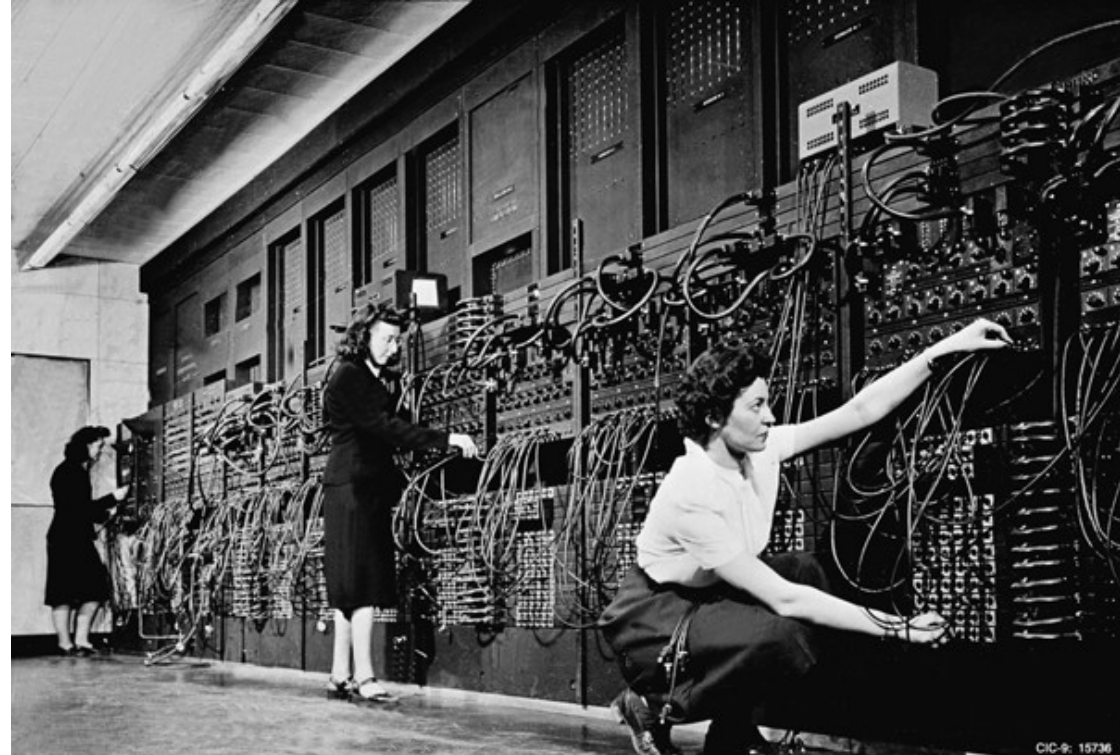# Software Engineering

1: Software Engineering

Definitions

- For clarity, we will use the following terms differentiate two different but related concepts

  - There is an overlap in concepts between the two

  - The distinction is primarily based on the purpose of the code development

- *Programming*

  - Writing and designing code in a specific programming language

- *Software Engineering*

  - The process of designing and building a software system using standard accepted engineering principles and practices

  - Independent of programming language

  - Essential when systems scale up

Before Software Engineering

- The early years (1940-50s)

- Hardware rules.

  - The photo shows how the ENIAC was programmed by interconnecting the electron tube registers with cables inserted in plug boards.

  - Computers were very expensive, very unreliable and primarily used as automated calculation engines.

  - Operating systems were in their infancy or non-existent so the primary role of software was to manipulate hardware, at least in those rare cases were software existed.

Before Software Engineering

- The Code Cowboys: 1950s -1970s

    – In the 1950s, computers continued to be very expensive

    – The computer pictured above is a Burroughs B55000 in 1964 which cost 50 million dollars in today's money. The computing power is less than than a cell phone today.

    – Hardware was the primary cost of computing, the quality of software was determined by how efficiently it used the hardware even if the code was unreadable by humans.

    – Programs were very small in size, written in assembler for efficiency and usually automated one particular calculation.

    – The primary role of computers was to automate repetitive, difficult and error prone math calculations

Before Software Engineering

- The mainframe era – 1970s - 1990s

  - Operating systems created a virtual layer between the programmer and the hardware that allowed programmers use high-level languages and focus on the problem domain rather than manipulating the hardware.

  - The problems being computerized where often automating existing systems rather than calculations

  - Users of the system expected the systems to work to their specifications

  - Automating these systems was a much more complex problem than doing rote calculations

  - Shown in the picture is an IBM 370 CPU being unpacked.

The Development Crisis

- Ad hoc programming works for a simple programming

  - You just write code and it works

  - But sort of process fails as the code base scales up in complexity and size

  - Continued failures in automation projects became a software development crisis

  - The solution was apply engineering techniques to software development

  - The quote from Harvey Ditel describes this realization

Many systems were being developed during the 1960s.

They were huge conglomerations of software written by people who really didn't understand that software, as well as hardware, had to be engineered to be reliable, understandable and maintainable.

Endless hours and countless dollars were spent tracking bugs that should never have entered the system in the first place.

Errors in the earliest phases of the project were not located until long after the products were delivered to the customers. These errors were enormously expensive to correct.

People turnover on the projects often resulted in large numbers of software modules being scrapped and then being rewritten by new people because the existing modules couldn't be understood.

So much attention was given to these problems that eventually computer scientists and industry people began devoting considerable resources to the problem of constructing computer systems.

This spawned the development of the field of software engineering.

The NATO Conference

In 1968 a group of computer scientists gathered for what became known as the NATO Software Engineering Conference.

The conference theme:

- The world is becoming more reliant on software

- Yet software systems are getting larger, miss deadlines, go over budget, and are brittle to change.

- Identify the factors that contribute to this reality, and propose ways to align the creation of software to the engineering discipline

- The photo is from the conference

The NATO Conference

- The solution was to adopt standard engineering practices and processes and adapt them to the construction of software.

  - Involved a variety of standard engineering practices like
    - quality assurance
    - requirements management
    - design processes
    - standardized construction techniques.

- This allowed for an explosion of computerization to take place from the 1970s to present time.

- However, the tools and techniques of software engineering be modified and adapted to meet the unique challenges of software development.

  - Same way engineering practices are modified for different types of engineering, civil versus aerospace engineering for example

- But no matter what "customization" of software engineering is used

  - They all share the same body of engineering best practices, processes and concepts.
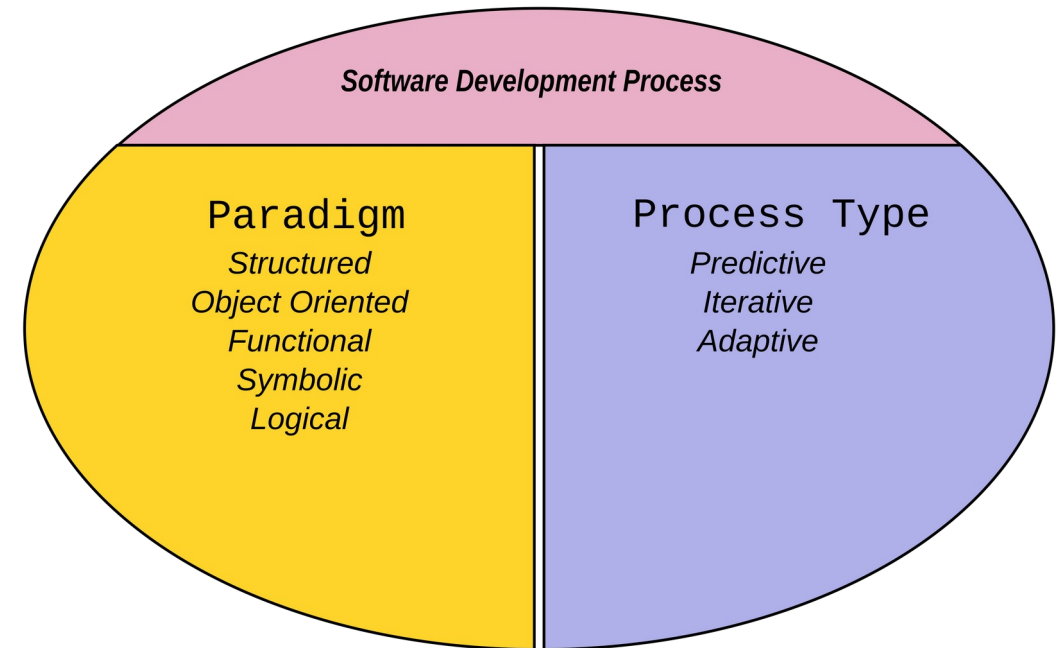
The Iron Triangle

- We often need to deliver software under the following conditions

  - Spec: It has to work the way the stakeholders have specified

  - Budget: There is a limit to the resources we can use to build the software

  - Timeline: We have delivery schedules to meet

- Not all programming is software engineering

- But when the iron triangle is in effect, we need to use an engineering methodology

To Spec

Within Budget

On Schedule
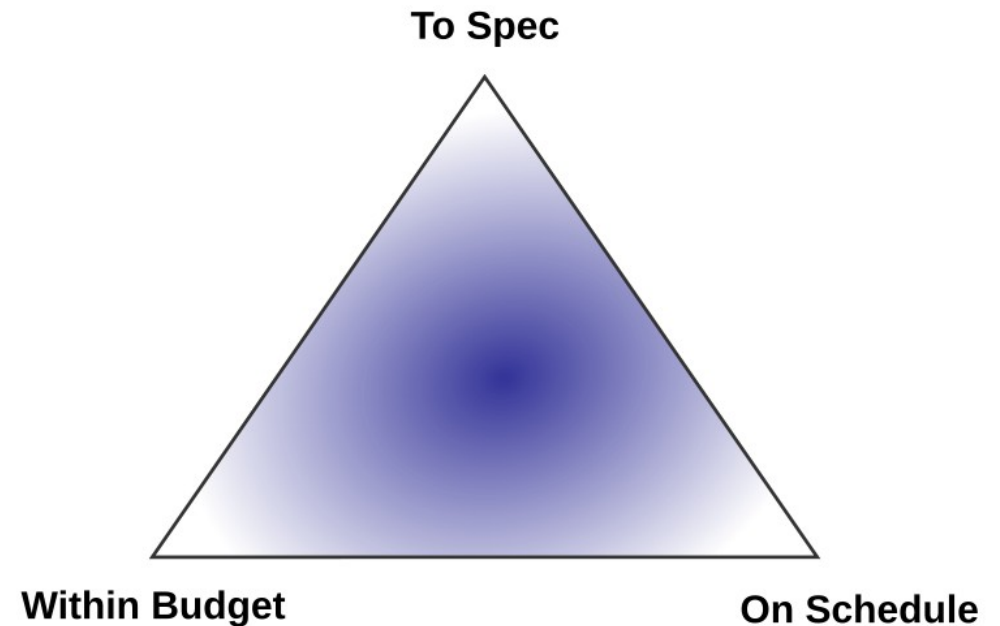
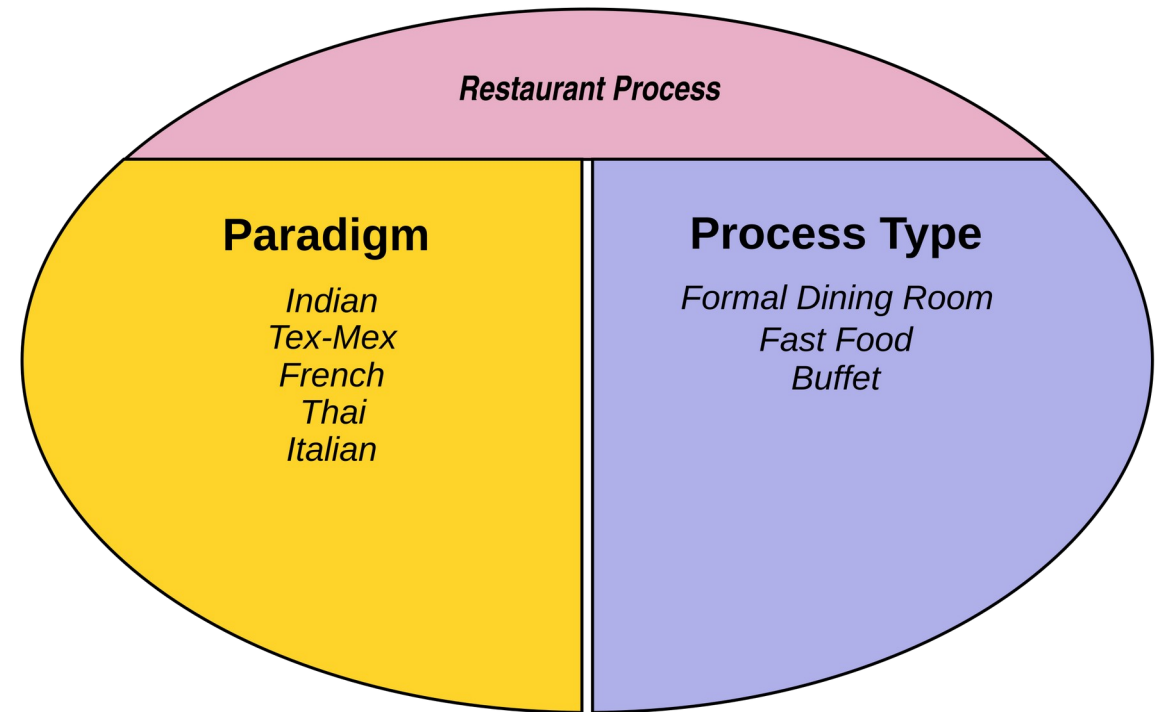*The **IRON** Triangle of Software Development*

Process and Paradigm

- Software development processes are made up of two parts:

  - A paradigm, which describes how to develop programs using a specific programming language or model

  - A development life cycle, which describes how to structure the paradigm activities in order to deliver the triangle under the constraints of the iron triangle

- Paradigms are defined generally as

  - The entire constellation of beliefs, values, techniques and so on shared by the members of a given community.

  - It also denotes one sort of element in that constellation, the concrete puzzle-solutions which, employed as models or examples, can replace explicit rules as a basis for the solution of the remaining puzzles of normal science

Software Development Process

Paradigm
Structured
Object Oriented
Functional
Symbolic
Logical

Process Type
Predictive
Iterative
Adaptive

Process and Paradigm

- We can see this distinction in many different areas where we "make things"

- When we look at restaurants we see:

  - Culinary Paradigms: TexMex, Cajan, French, etc

  - Food Production Life Cycle: Fast Food, Fine Dining, Buffet, Take Out, etc.

- The paradigm describes how to cook the food

- The process is necessary for restaurants to meet the iron triangle

- We can think of restaurants as applying engineering principles to food delivery

Restaurant Process

Paradigm

Indian
Tex-Mex
French
Thai
Italian

Process Type

Formal Dining Room
Fast Food
Buffet

The Engineering Cycle

The engineering process is a description of the logical sequence of natural questions or problems as they occur in any kind of construction process.

The process is universal and can be observed any time we are successfully building or delivering something in an engineering or business context.

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Requirements

The basic question that has to be answered during requirements is *"Why are we building this?"*
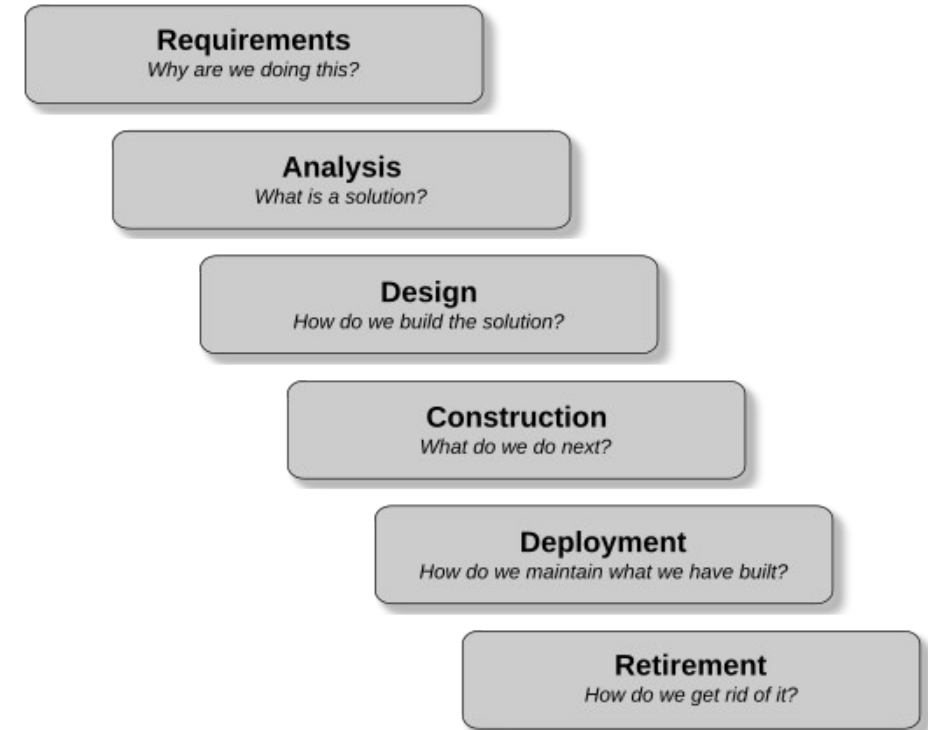
- This is often called the "value proposition"

- Identification of the problem to be rectified

This is generally followed by two other kinds of questions.

"What do the users need what we are building to do for them?"

- At this stage, we need to know the functional requirements (what the stakeholders expect what we are building to do for them)

- And the non-functional requirements (what the performance criteria are that what we are building has to meet).

"What are the constraints – budget, time, architecture, etc. – that have to be considered when building the solution?"

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

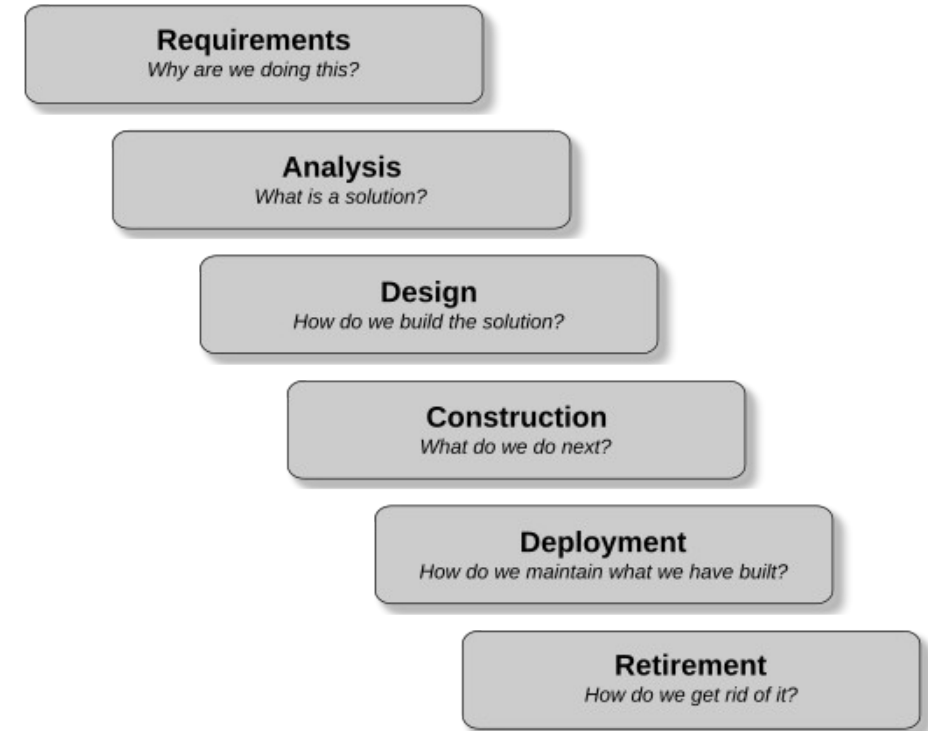**Retirement**
How do we get rid of it?

Requirements

It is not always possible to identify all of the requirements at the beginning of the project

We may need to use some adaptive methodology like an Agile approach

Iterations help validate assumptions about the underlying problem

And help identify erroneous assumptions about what the requirements are

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?
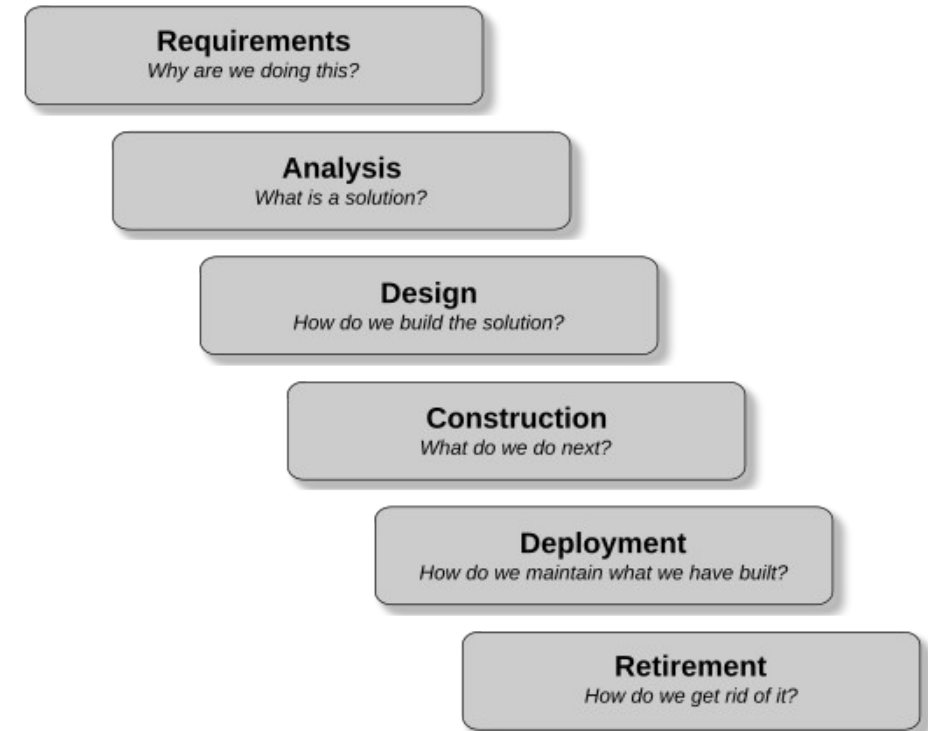
Requirements

"Go the the source of the problem"

Find out what the real requirements are first hand from the people directly who have them

Requirements of often supplied second hand

- "This is what the people I supervise need."

- Often "This is what I think my people need."

Identify stakeholders

- Groups that have requirements

- And the capability to force you to adopt them

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

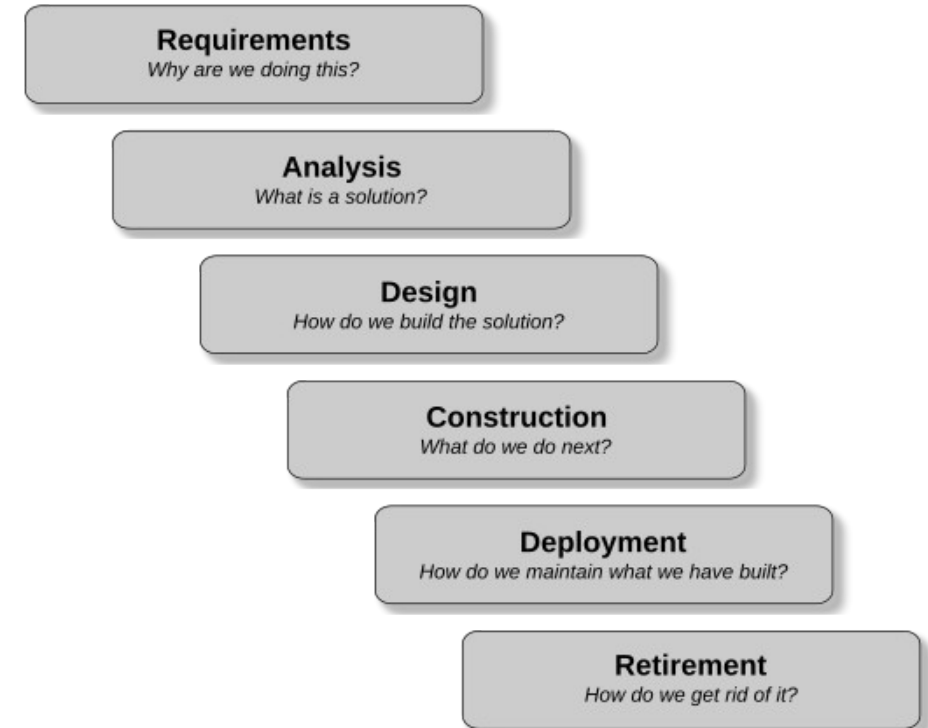Retirement
How do we get rid of it?

Requirements

Primary stakeholders

- Impacted directly by the project

- Customers, staff, process owners, data owners budget controllers

- Have project specific requirements we can ask them about

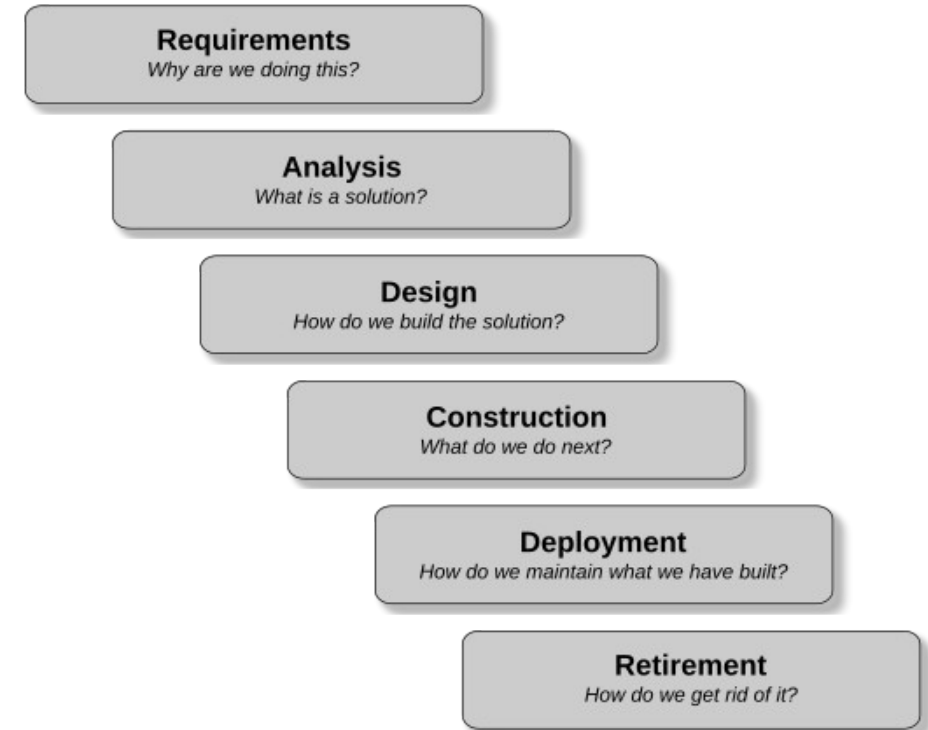## Secondary stakeholders usually regulate a domain of activity

- Data governance, accessibility standards

- Do not have project specific requirements

- Their requirements are usually published

- Non-responsive to inquires ("Go look it up in our publication on the topic")

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Requirements

Secondary stakeholders

- Often the source of constraints

- IT infrastructure standards

- Legal and regulatory compliance

- SLAs for that type of project

- Business case requirements

- Security requirements

- Audit and accountability requirements

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

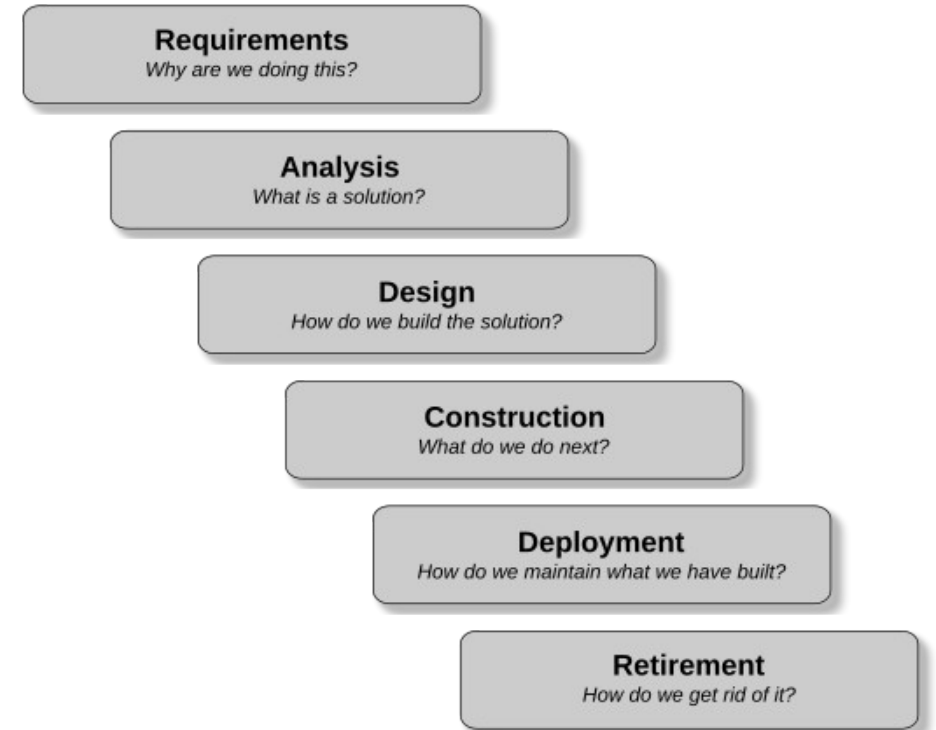**Retirement**
How do we get rid of it?

Analysis

In the analysis phase, we find a solution to the problem.

- We often formulate the solution in a specification that describes what is to be built.

- The specification serves as a design target

Whatever design we propose has to produce a product that meets the description of the solution in the specification.

- We may come up with multiple ways to solve the problem which differ in terms of cost, quality and other factors

- But eventually, we need to settle on a single solution.

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

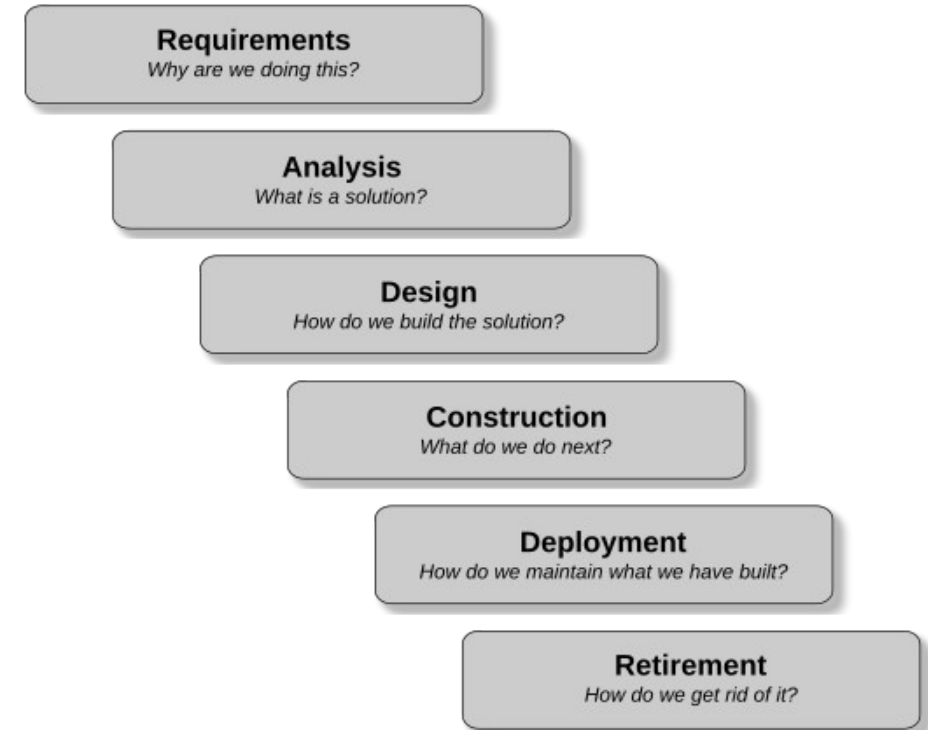**Retirement**
How do we get rid of it?

Analysis

If there are multiple possible solutions then we may need to experiment

- Small pilots or proof of concept

- Experiment with different architectures

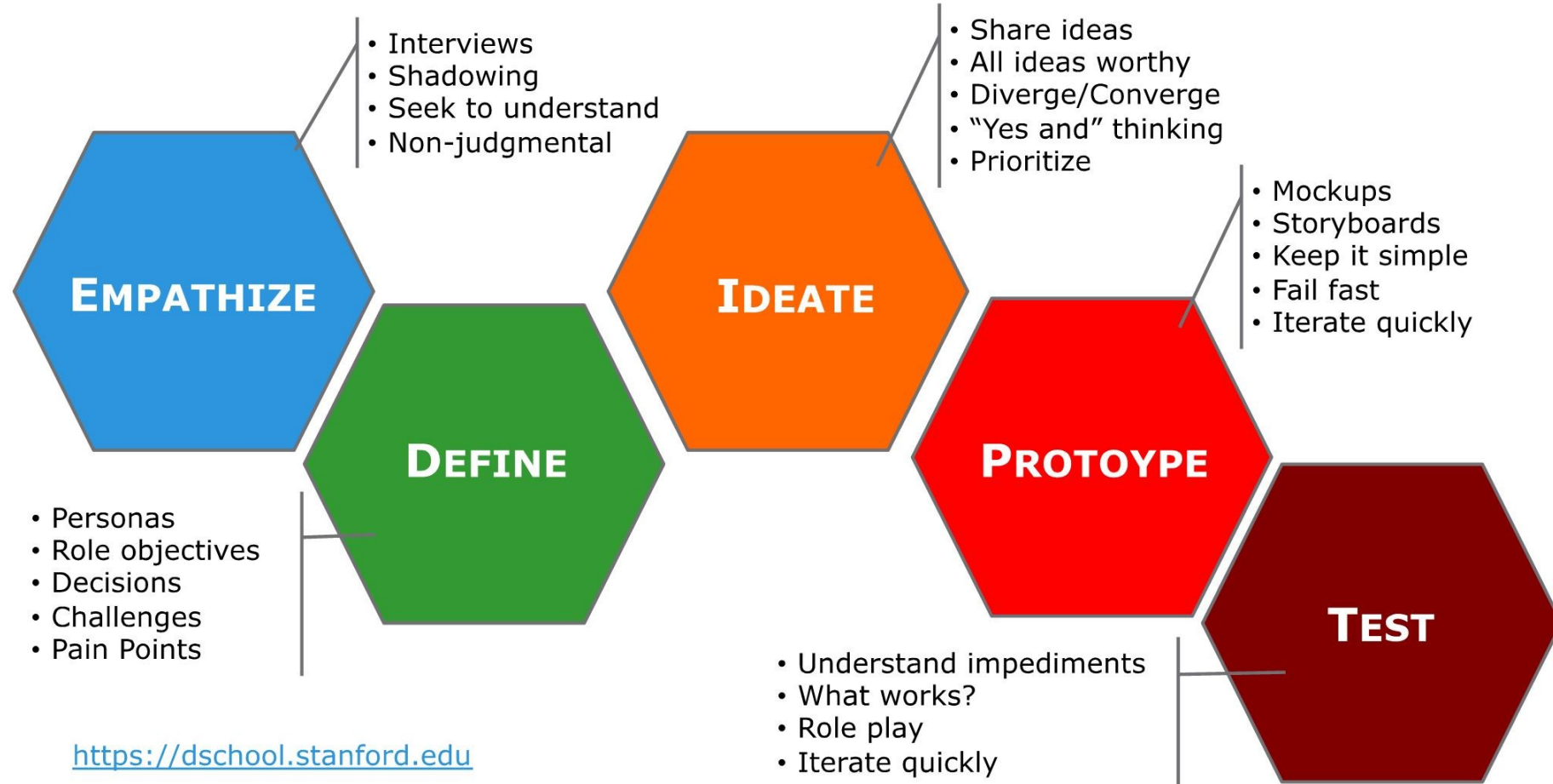- Experiment with different solutions

The goal of this experimentation is to predict how viable our proposed solution would be

- This is an example of writing code that is not under the iron triangle

- Often a quick and dirty prototype

- This sort of approach is often done in a frame work like the Design Thinking methodology for new product development

Requirements
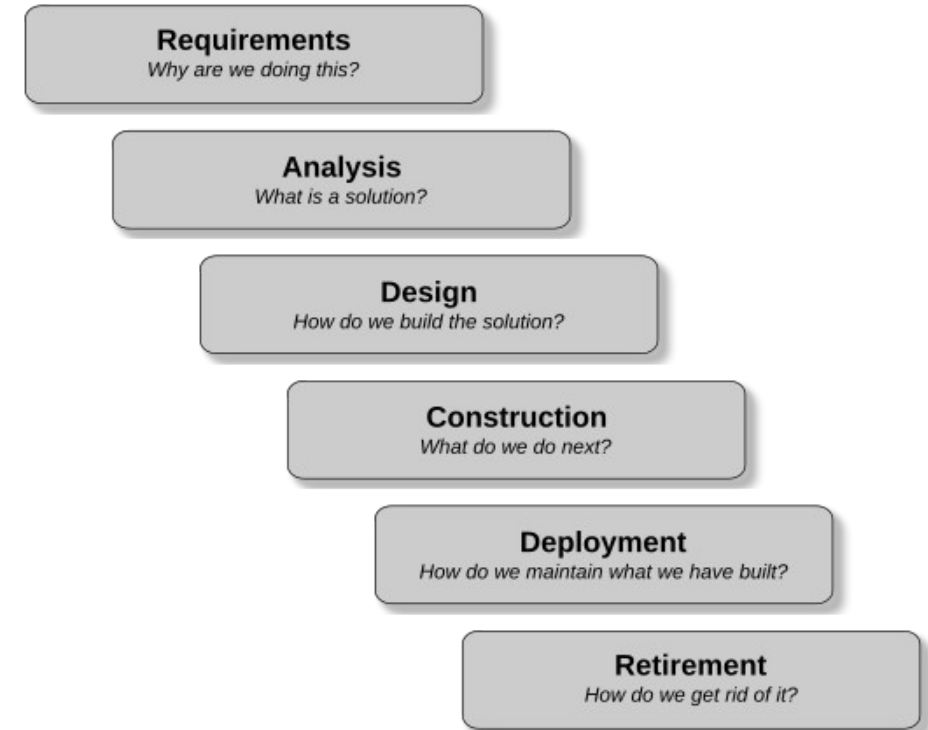*Why are we doing this?*

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Design Thinking

Stanford d.school Design Thinking Process

EMPATHIZE
- Interviews
- Shadowing
- Seek to understand
- Non-judgmental

DEFINE
- Personas
- Role objectives
- Decisions
- Challenges
- Pain Points

IDEATE
- Share ideas
- All ideas worthy
- Diverge/Converge
- "Yes and" thinking
- Prioritize

PROTOYPE
- Mockups
- Storyboards
- Keep it simple
- Fail fast
- Iterate quickly

TEST
- Understand impediments
- What works?
- Role play
- Iterate quickly

https://dschool.stanford.edu

Analysis

At some point, we may need to make a "no go" decisions

- We may find that we are not the improvements we expected

- We may not have the infrastructure support we need to sustain deployment

- We may not be able to use the data we need

- There may be aspects of the problem we were unaware of prior to starting the project

- We may not have the organizational support for the project

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

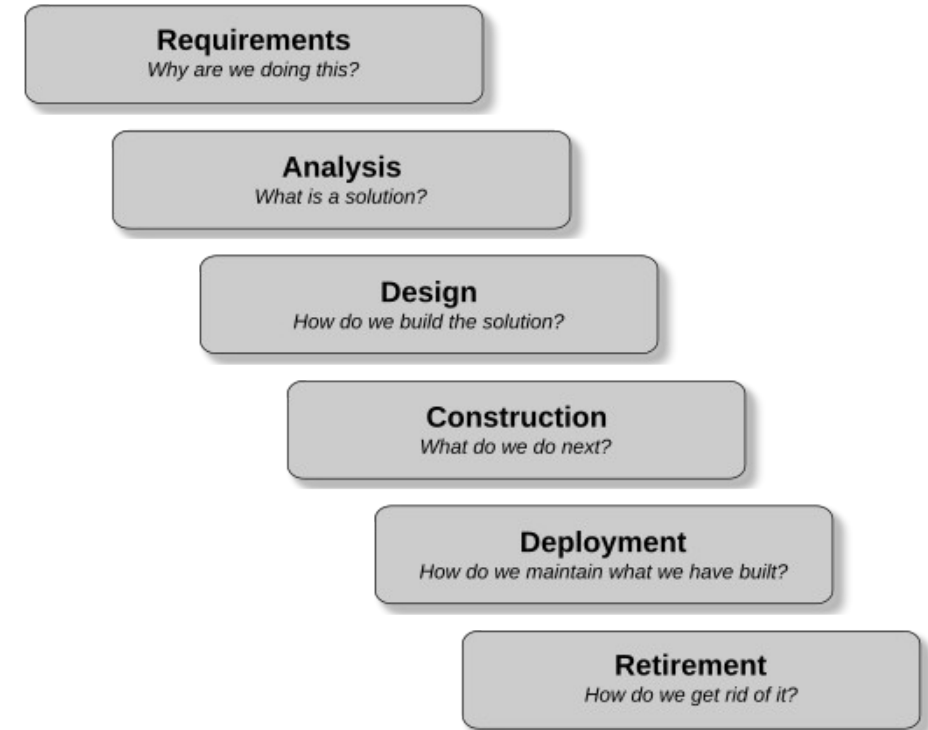**Retirement**
How do we get rid of it?

Design

A specification describes a solution but does not tell us how that solution should be built.

- A design describes how the solution is to be built given a specific set of resources, capabilities and constraints.

- For a single specification, we may have a number of different designs depending on what we have available to work with.

Early prototypes during Analysis

- Will have an informal high-level design

- Lightweight – more of a plan for experimentation

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
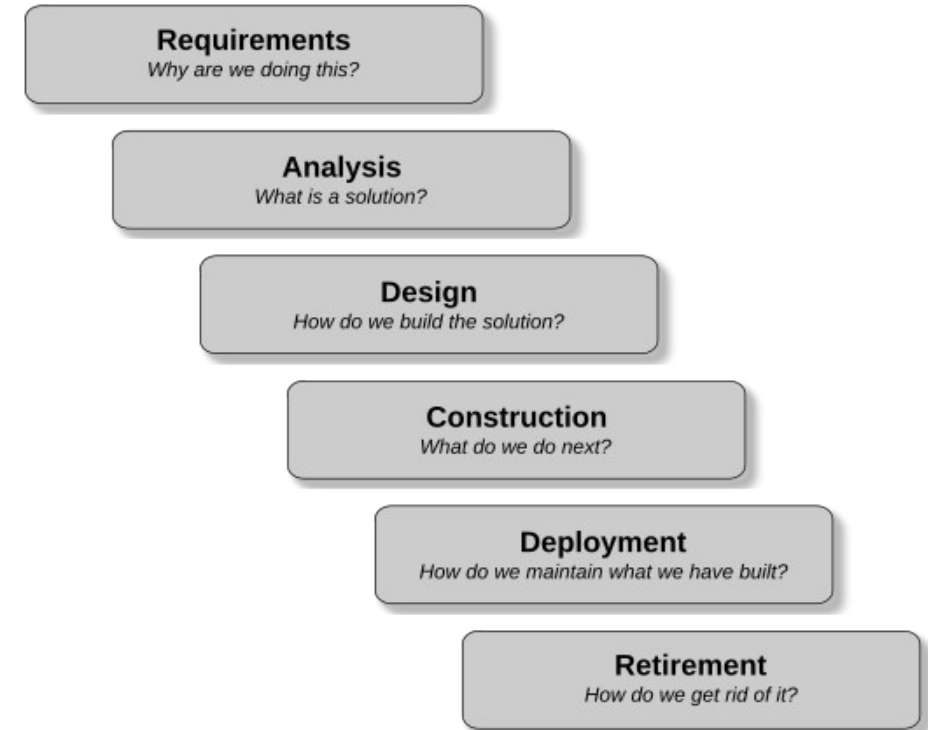*How do we get rid of it?*

Design

A Design:

- Can be thought of as the project plan

- Defines an application architecture

- Defines the components

- Defines the QA requirements and release baselines

DevOps approach

- We also define the automation CICD pipelines that will be used

- We design the development and operations infrastructure as part of the design

- Includes the monitoring, continuous evaluation and feedback

- We will return to this idea later in the course

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

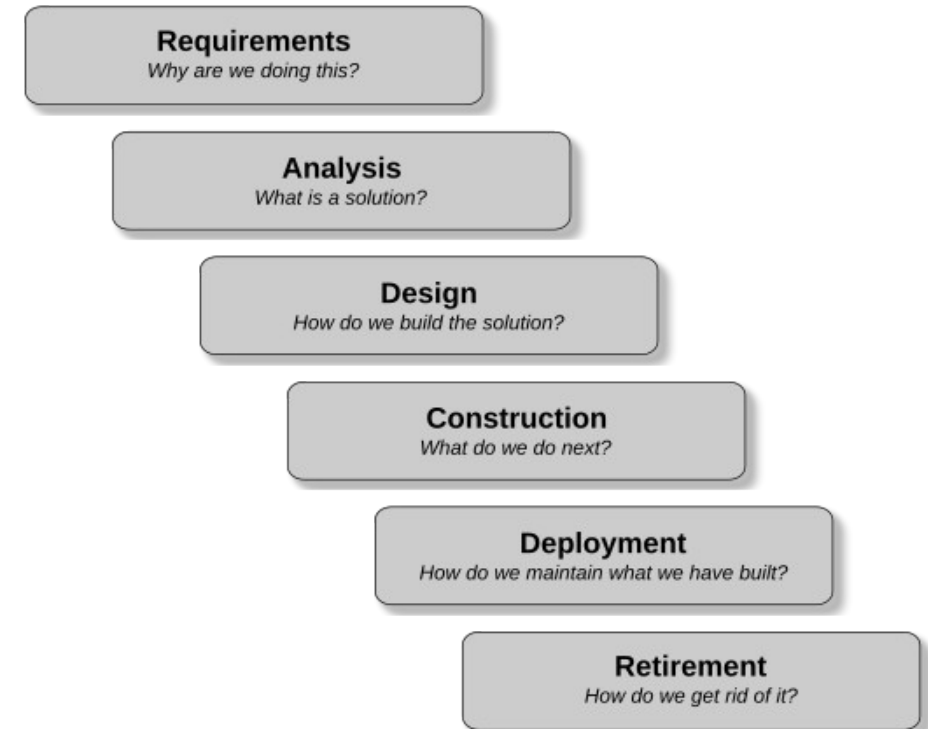**Retirement**
How do we get rid of it?

Construction

The product described in the specification is built according to a given design and construction or project plan.

- Having a design alone is not enough for something to be built,

- Especially as the size and complexity of what you are building increases.

- We may also need to build the development infrastructure

You also have to use the appropriate construction techniques and plan out the development activities.

- This is where the actual programming takes place

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

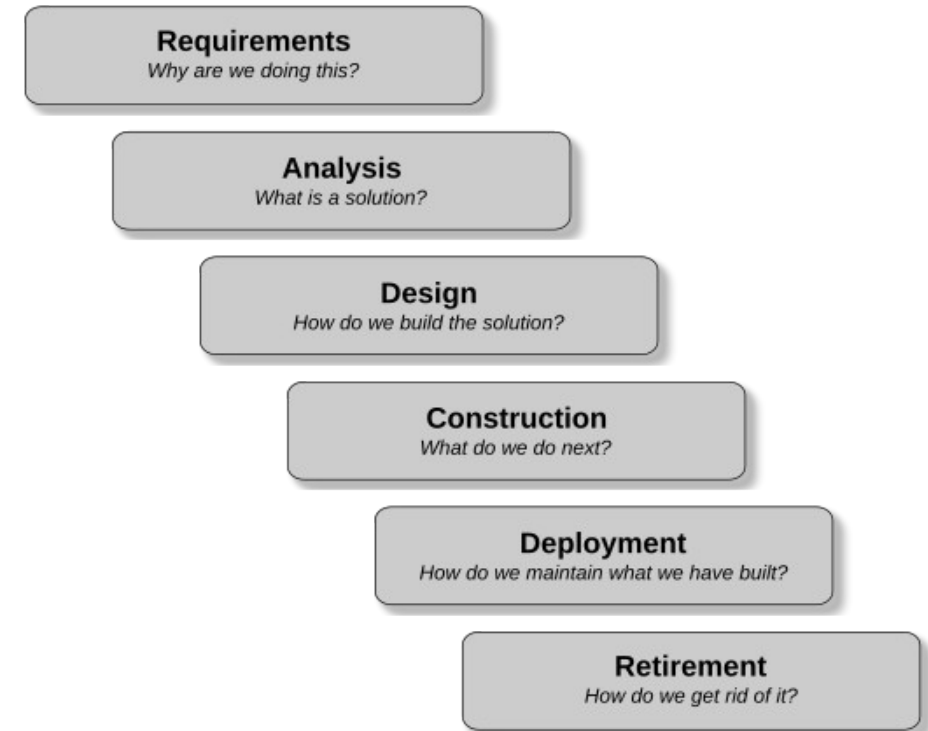**Retirement**
How do we get rid of it?

Deployment

The completed solution is deployed when it goes into use.

The questions that we have to ask for the deployment phase deal with might be called operational issues such as:

- "How are changes or repairs to be made?"

- "How are users and maintenance people to be trained?"

- "How do monitor the product for possible malfunctions?"

- "How do we monitor the level of performance in terms of expected SLAs?"

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?
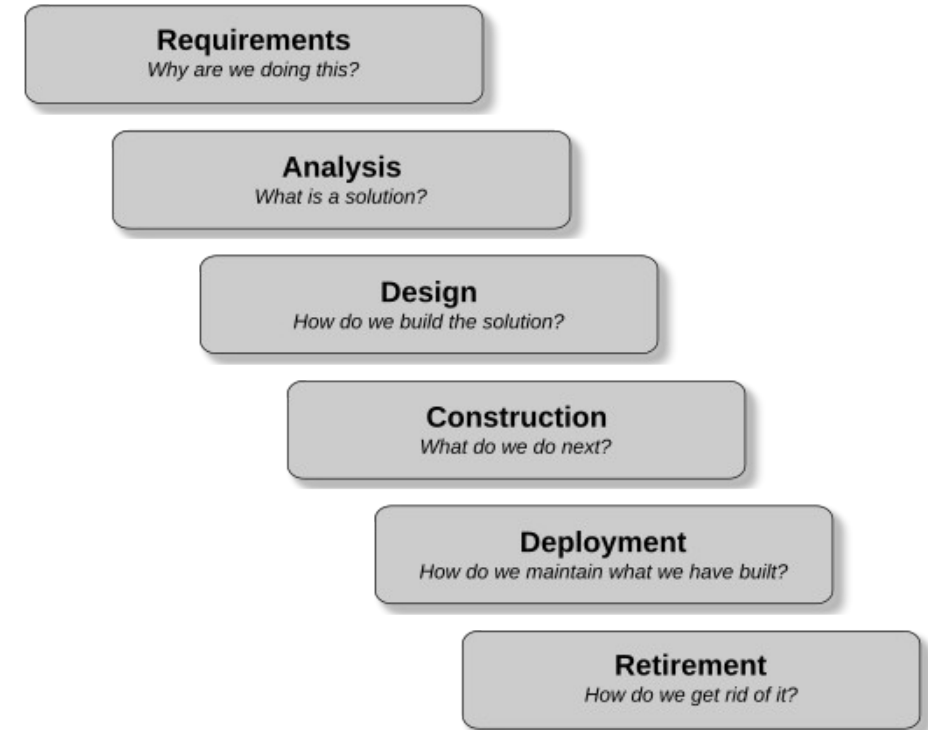
Deployment

If we are using a DevOps process

- We will use some sort of staged deployment

- In both production and prototype

A number of basic types

- Canary deployment

- Blue/Green deployment

- More detail later in the course on these methods

Preferably automated

- Each allows a roll-back path

- In case the deployment goes bad, we can immediately rollback to the previous deployment

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Deployment - Monitoring

Prior to deployment, KPIs and SLAs need to be established

- KPI – Key Performance Indicator

- SLA – Service Level Agreements
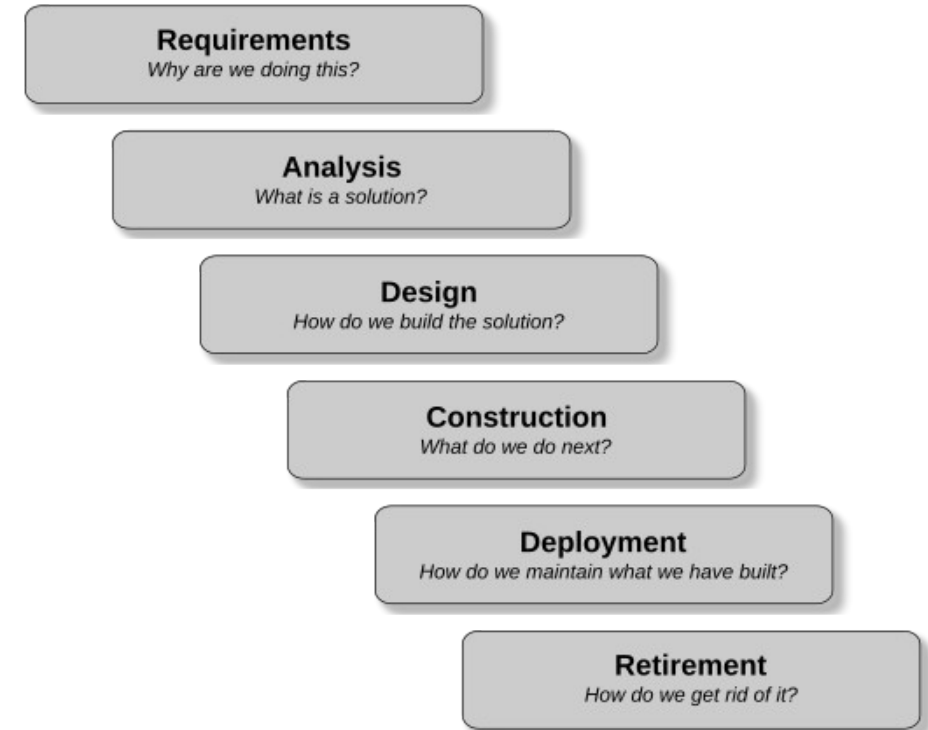
Some of these are performance based

- Often collected and evaluated with automated tools

Some of these are acceptance based

- Requires gathering feedback from users of the application

Some of these are correctness or value based

- Are we getting the right results?
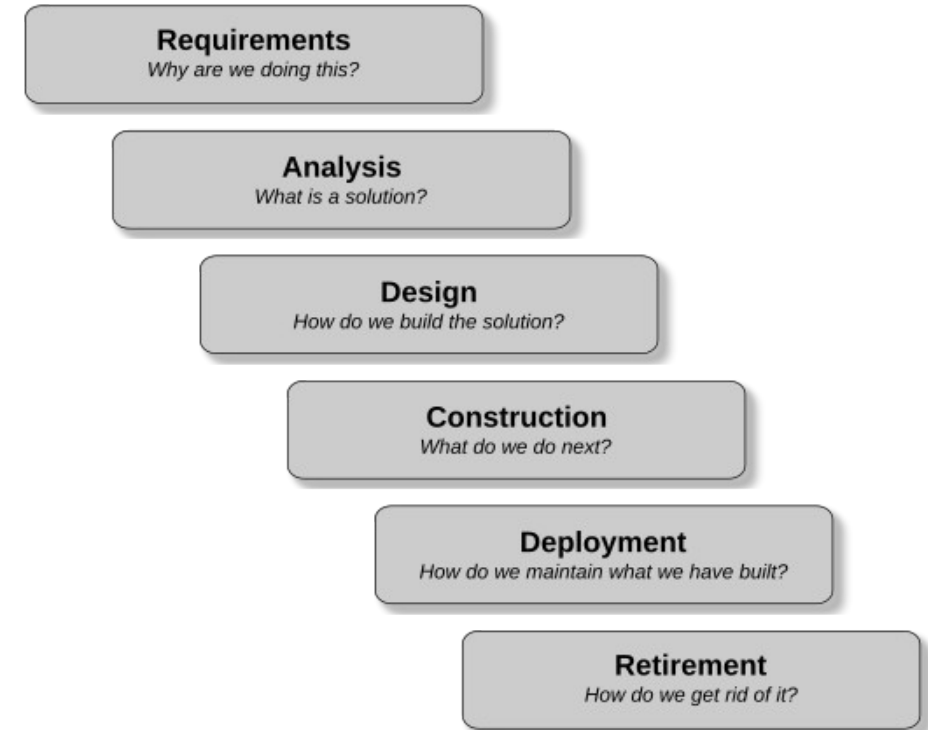
- May require a manual review

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Retirement

Retiring the application is a business decision

- Do we replace it or upgrade it?

The design should have include a retirement plan

- For example: An "un-rollout" strategy for a staged phase out of the exsiting system

- Including a potential migration path for data and other artifacts to another system

- Preservation of whatever is needed for records compliance or legal compliance

Failure to do this in the past has resulted in a mass of legacy systems that are extremely difficult to replace

Requirements
Why are we doing this?

Analysis
What is a solution?

Design
How do we build the solution?

Construction
What do we do next?

Deployment
How do we maintain what we have built?

Retirement
How do we get rid of it?

Legacy Monolithic Issues

Codebase is enormous

- Little or no documentation

- Coupled to legacy technology

Eg. Internal Revenue Service

- Running 1960s vintage code

- Application highly complex

- Attempts to replace it have failed

- Over $15 billion so far with minimal succes

The IRS is not unique

# Requirements, Specifications and Designs

- Defining requirements, specification and design.

    – These terms are not used consistently, unambiguously or clearly.

    – For this course, we will use each of these terms to mean one thing and only one thing

- A *requirement* is a condition or capability needed by a stakeholder to solve a problem or achieve an objective.

    – "I need to be able to to find all the invoices associated with a customer, and I don't care how you do it, just do it."

- A *feature* is a condition or capability possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document.

    – There is a "search for invoices" button on the customer information screen.

- A *specification* (or feature description) is a documented representation of a feature

Requirements, Specifications and Designs

- Requirements describe the needs, wants, wishes and desires of the stakeholders.

  - Requirements do not describe the appearance or behavior of the system

  - Each requirement may have a number of possible solutions

- Specifications describe the solution - the system to be built.

  - It is a design target and a basis for discussion between the stakeholders and developers about what the system should look like and what is should do.

  - It also serves as a baseline for testing and quality assurance.

  - The solution described in the specification should satisfy the requirements of the stakeholders as much as it feasible to do so within resource constraints (iron triangle).

# Requirements, Specifications and Designs

- Designs are blueprints that describe how to build what the specification describes.

  - A design describes how the to build the solution *given a specific set of resources*.

  - This means that a single specification may be implemented by a range of designs.

  - Depends on factors like available skill sets, technology choices and system architectural constraints

ProTech
protechtraining.com
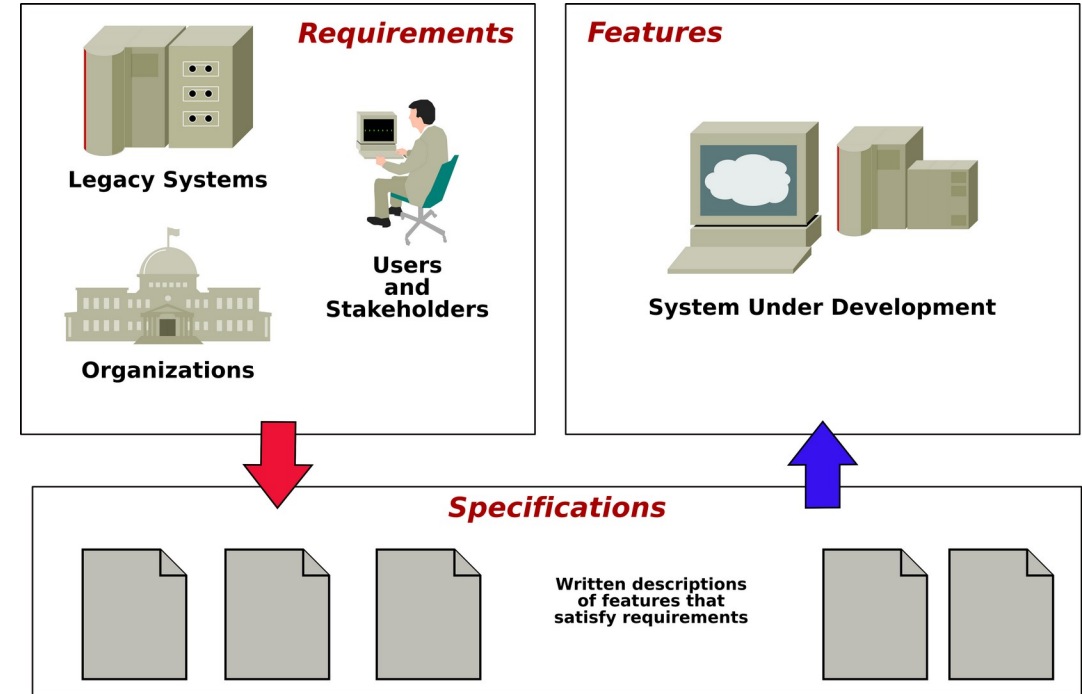
The Generic Software Process

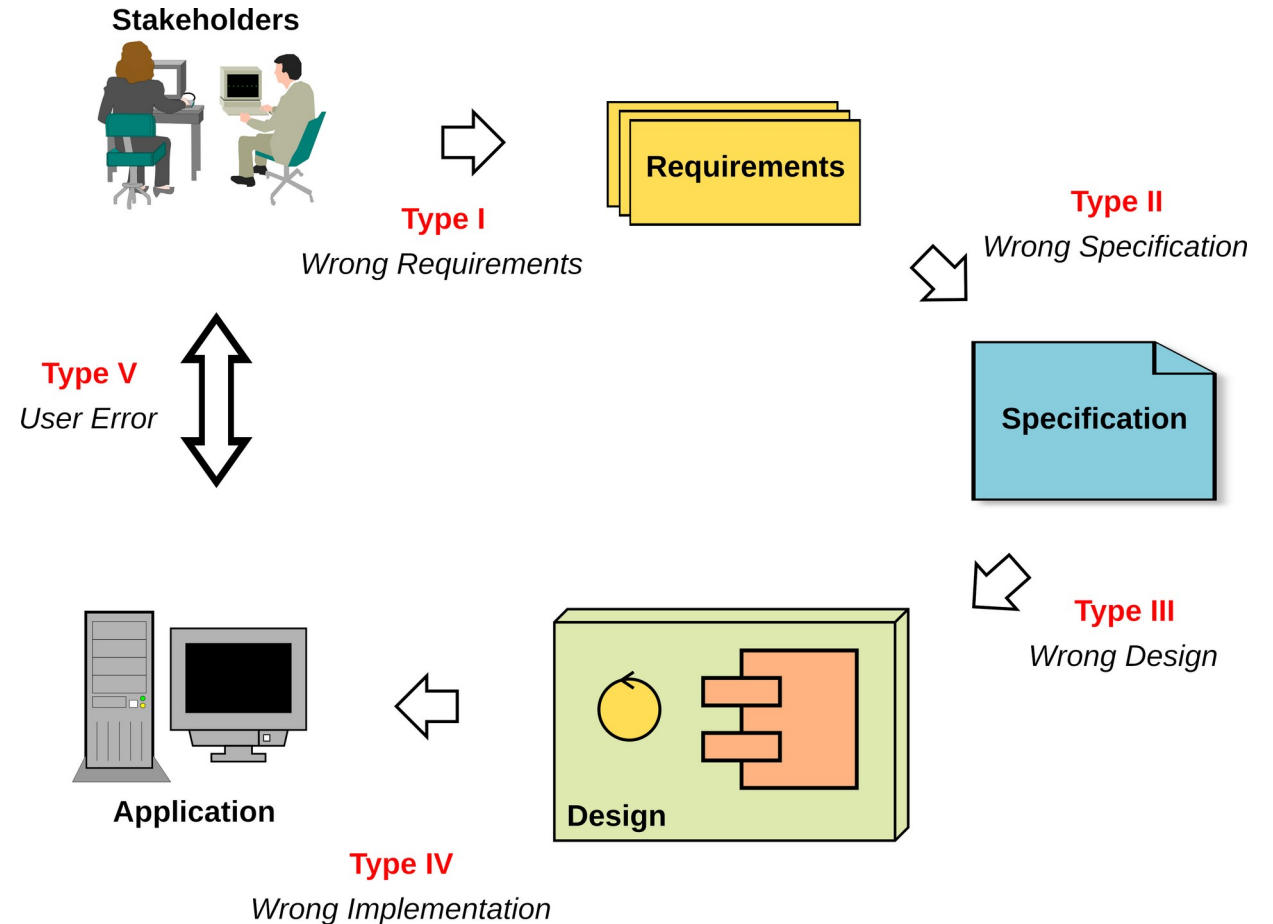Most of the standard representations of the generic software development process are similar to the diagram

- Traditionally, the deployment and retirement phases are ignored

- DevOps does rectify this to a degree

- Even Agile methodologies often only deal with the phases shown, not the operations and retirement phases

Stakeholders

Gather

Requirements

Solve

Specification

Use

Plan

Application

Build

Design
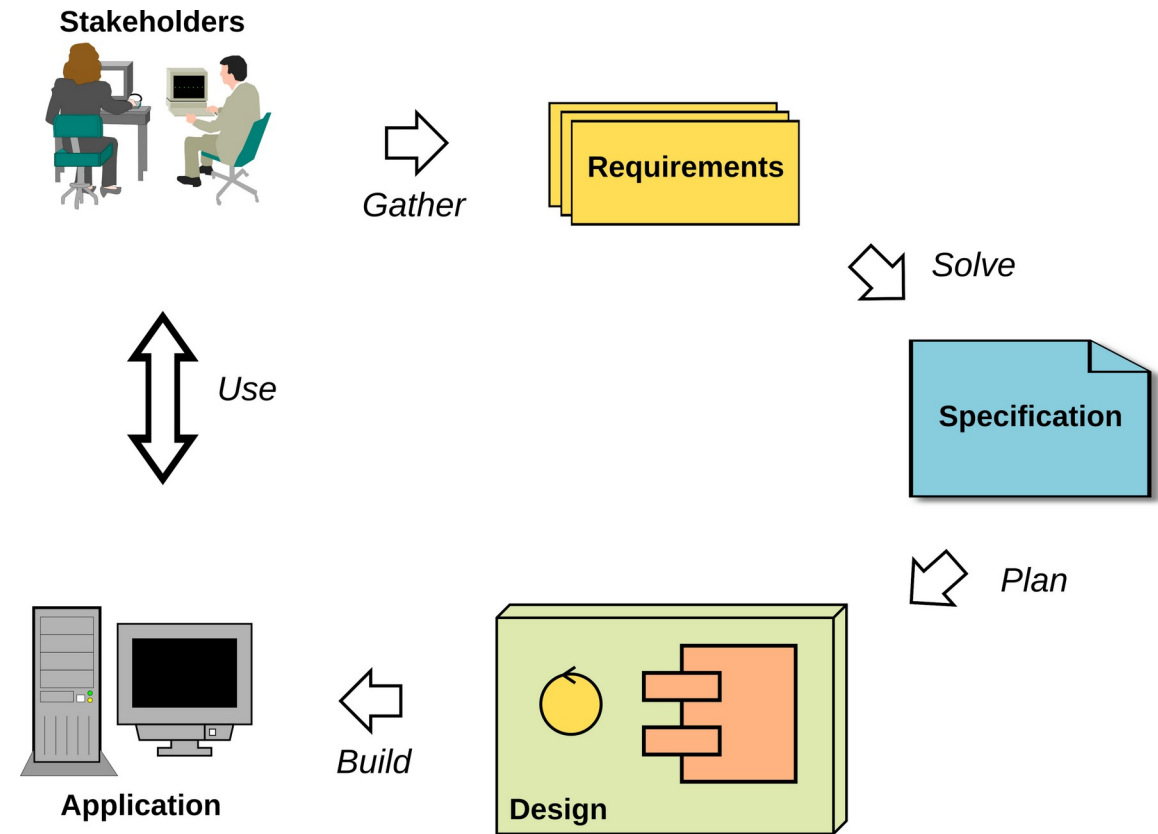
Where Things Go Wrong

Type I: Requirements are wrong.

- This is the most difficult kind of error to correct since it cannot be detected within the project but can only be detected with the participation of the stakeholders who are outside the development effort.

- In the worst case scenario, identification of the error takes place after the finished product is deployed and being used by the stakeholders.

Stakeholders

Type I
Wrong Requirements

Requirements

Type II
Wrong Specification

Specification

Type V
User Error

Type III
Wrong Design

Application

Design
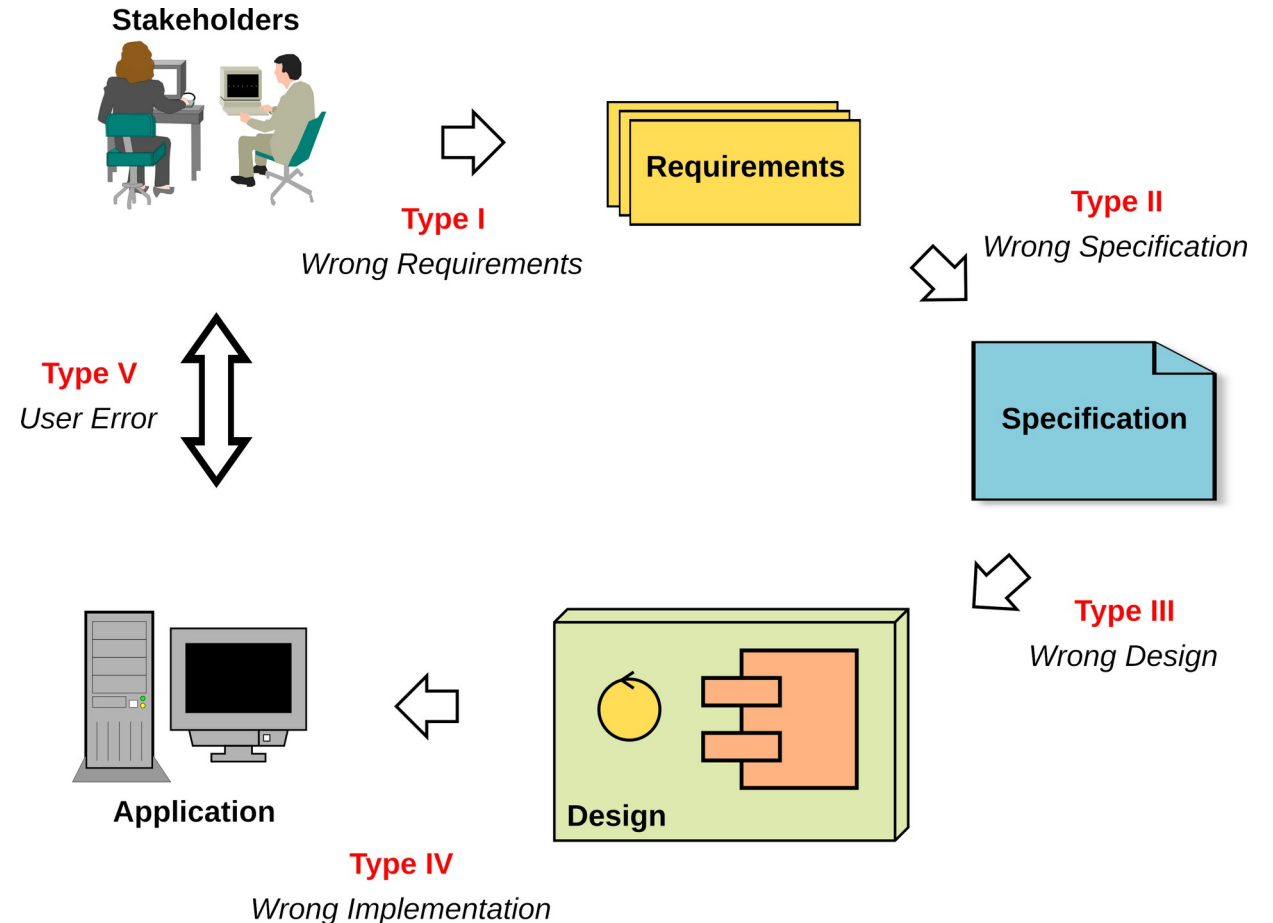
Type IV
Wrong Implementation

Where Things Go Wrong

Type II: Specifications do not satisfy the requirements.

- Our system may work perfectly, but it does the wrong thing perfectly.

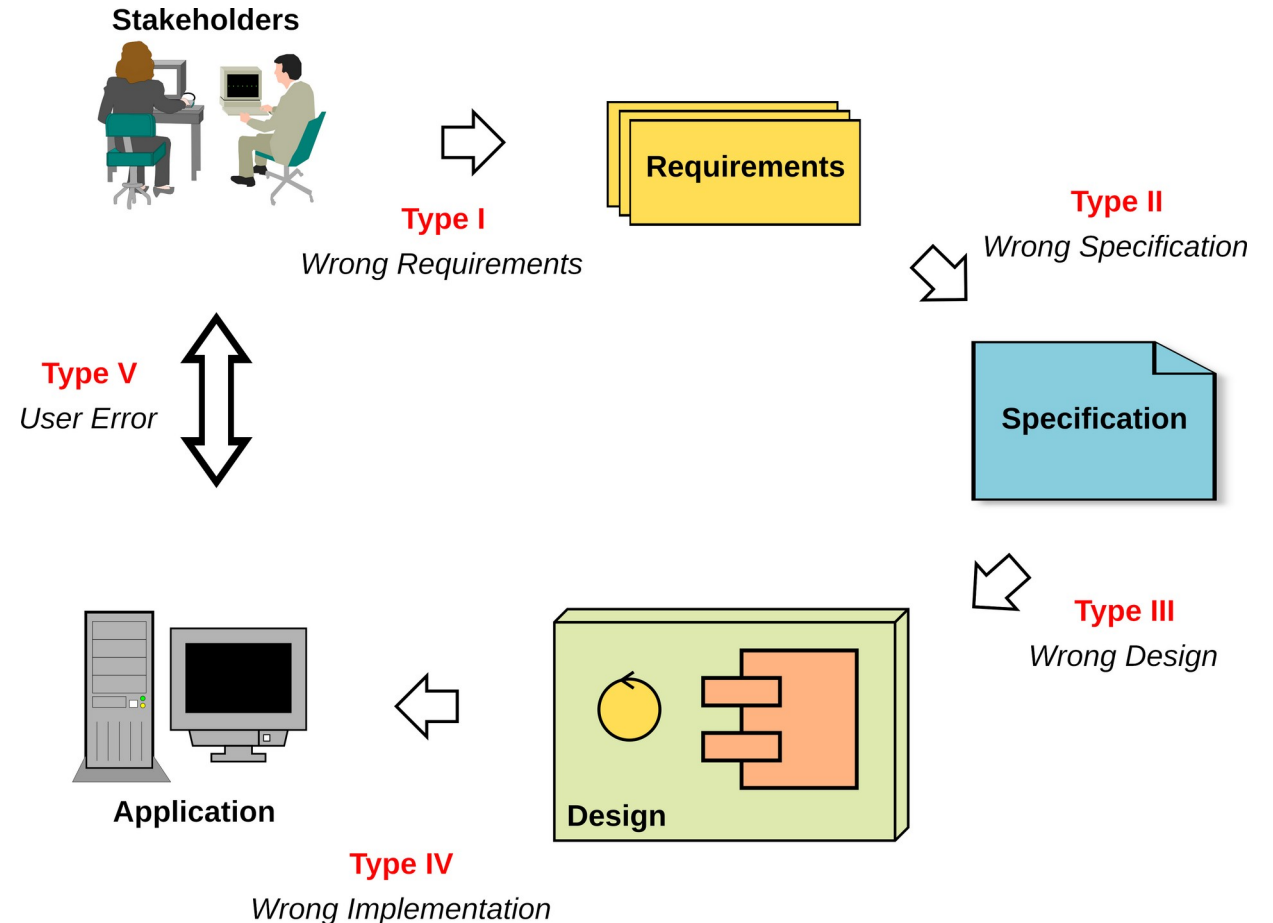Generally, a type II error occurs if there are:

- Requirements that are not addressed by the specification (omissions)

- Features provide functionality that does not meet any requirement (surprise functionality)

- A proposed feature that does not satisfy the requirement correctly (failures).

Stakeholders

Requirements

Type I
Wrong Requirements

Type II
Wrong Specification

Specification

Type V
User Error

Type III
Wrong Design

Application

Design

Type IV
Wrong Implementation

Where Things Go Wrong

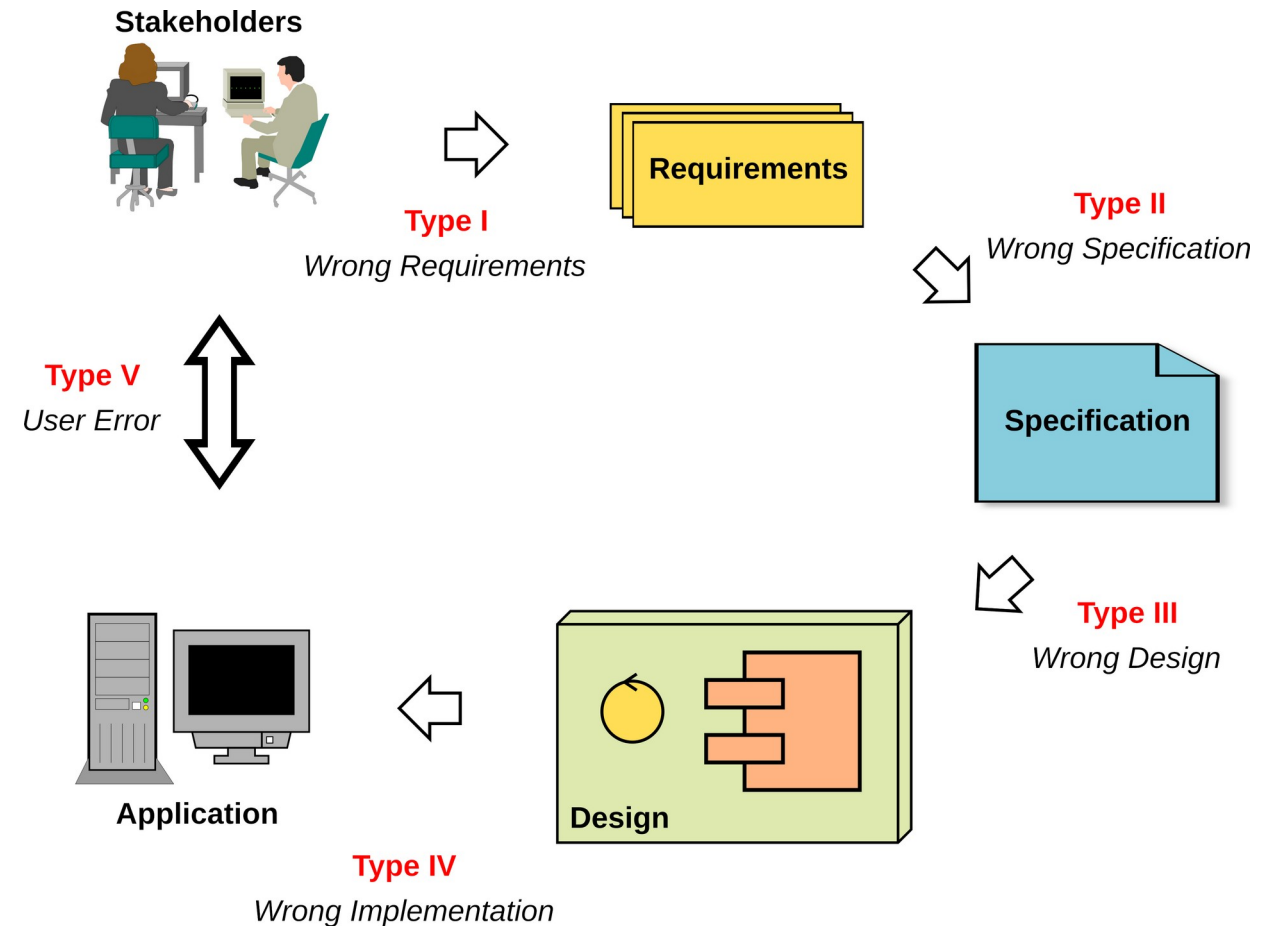Type III: System design does not meet the specifications.

- Following the design leads to a system that does not operate as specified.

- For example, the specifications may require the use of encryption for security, but the encryption methodologies were omitted in the design by mistake.

- Or the architecture proposed cannot handle the projected throughput or exceeds the allowable response time.

Stakeholders

Requirements

Type I
Wrong Requirements

Type II
Wrong Specification

Specification

Type V
User Error

Type III
Wrong Design

Application

Design

Type IV
Wrong Implementation

Where Things Go Wrong

Type IV: The system build and artifacts do not conform to the design.

- Someone made mistakes in building the system

- Whether writing code or performing some other construction task that were specified in the design

- Omitting exception handlers for example.

Stakeholders

Type I
Wrong Requirements

Requirements

Type II
Wrong Specification

Specification

Type V
User Error

Type III
Wrong Design

Application

Design

Type IV
Wrong Implementation

Where Things Go Wrong

Type V: User Error.

- This type of error is an unavoidable human error.

- The problem is that people do not always behave rationally and, while we may have designed and implemented everything correctly, it may be used in an irrational manner.

- It can also happen because users don't know how to use the system or they are using it in a different context that it was intended.
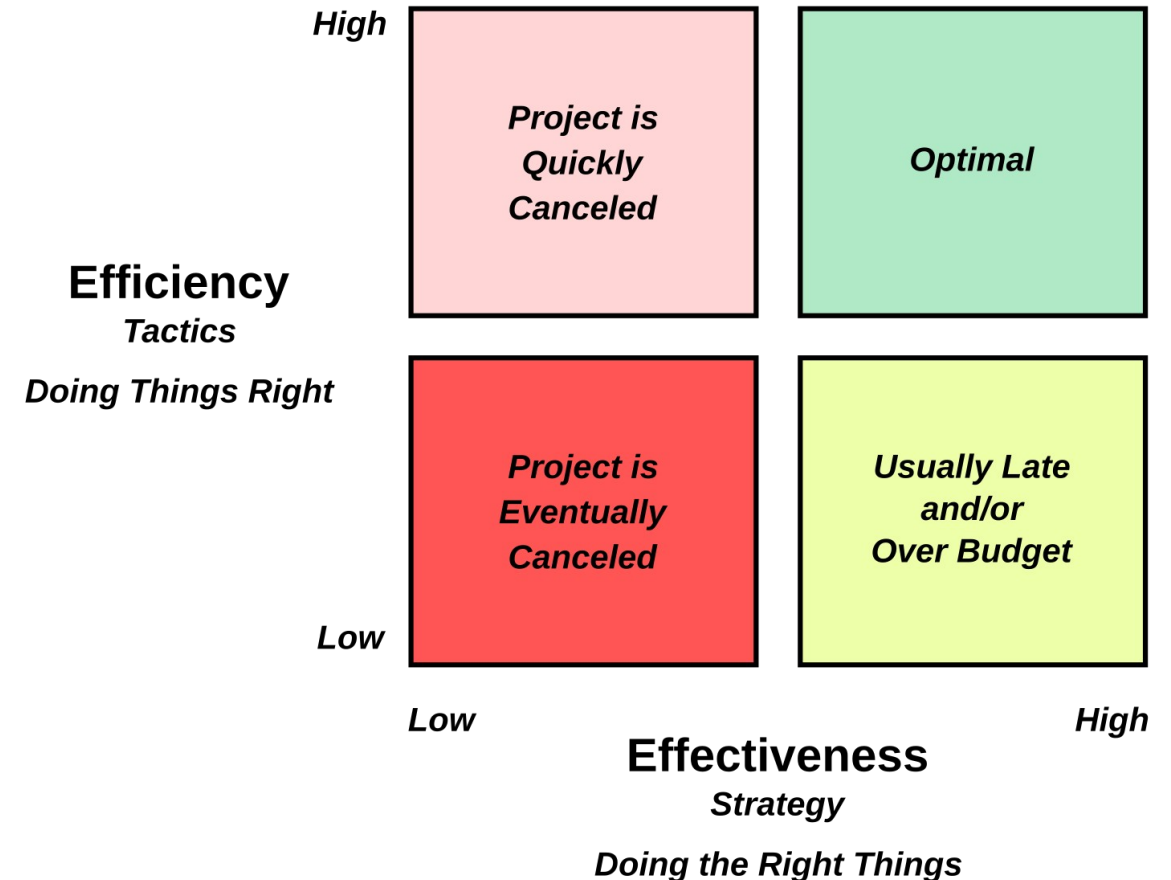
Efficiency and Effectiveness

Effectiveness:

- A process is effective when it produces the right result.

- We often call this delivering to spec - delivering software that meets the stakeholder needs and solves the business problems as it was supposed to.

- Effective software development processes build the right software.

Efficiency:

- A process is efficient when it is optimal in terms of how development resources are used.

- One of the main symptoms of a process not being efficient is how much rework is done during development.

- Rework is where previous work has to be done over because of errors made in the previous work.

- Efficient software development processes build software with the optimal use of resources.

Efficiency
Tactics

Doing Things Right

	High	
	Project is Quickly Canceled	*Optimal*
	Project is Eventually Canceled	*Usually Late and/or Over Budget*
	Low	
	Low	*High*

Effectiveness
Strategy

Doing the Right Things

Crushing the Iron Triangle

- The types of errors discussed in the previous section are exactly what have to be controlled for a software project to be successful

- One of the great computer scientists of the 20th century, Edsger Dijkstra commented:
    - *The most important single aspect of software development is to be clear about what you are trying to build.*
    - That suggests that a critical potential project point of failure happens when we are unsure of what we are building
    - If we don't know what we are supposed to be building, then we will never know if we are building the right thing.
    - However, the data suggests that projects often don't know what they supposed to be building.

The Chaos Report

Annual report from the Standish group.

Defines three resolutions for technical projects:

Resolution Type 1 – Successful Projects:

- The project is completed on-time and on-budget, with all features and functions as initially specified. Acceptance tests all pass.

Resolution Type 2 – Defective Projects:

- The project is completed and operational but over-budget and/or late and/or is delivered fewer features and functions than originally specified, usually with acceptance test failures.

Resolution Type 3 – Failed Projects:

- The project is canceled at some point during the development cycle.

Pie chart: Failed Projects 31%, Successful Projects 16%, Defective Projects 53%
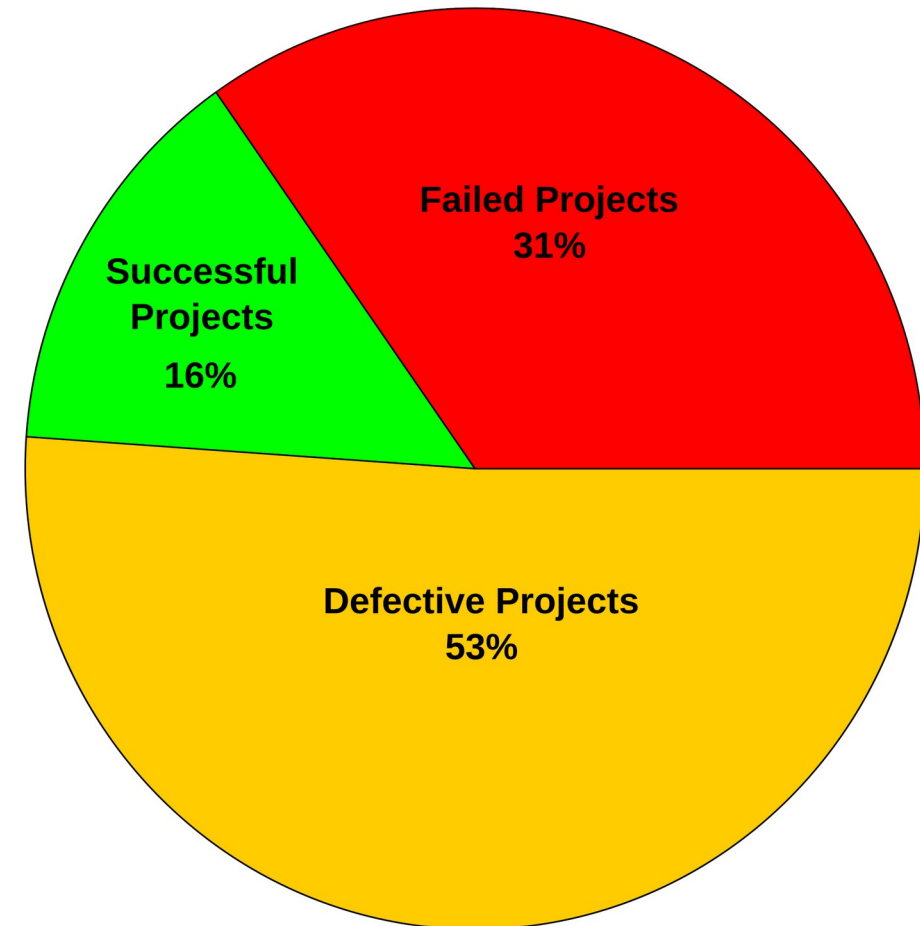
Successful Project Factors

The table lists the factors that were identified by people on the project as being responsible for a project being successful.

- The red italicized factors are those that are related to requirements and interactions with the users and stakeholders.

- Two of the top three items concern quality of requirements and getting the users involved.

- The fifth item, realistic expectations, suggest that the developers had early input into the project to help the business understand what could be done within the time and budget constraints of the project.

Success Factors	%
User Involvement	*16*
Executive Management Support	14
Clear Statement of Requirements	*13*
Proper Planning	*10*
Realistic Expectations	*8*
Smaller Project Milestones	8
Competent Staff	7
Ownership	5
Clear Vision and Objectives	*3*
Hard Working and Focused Staff	2
Other	13

Defective Project Factors

The table lists the factors that were identified by people on the project as contributing to a project being challenged or defective.

- By definition a challenged project is one that was completed but either missed its budget targets, delivery date or did not perform as originally specified.

- The top three reasons why projects were defective or challenged have to do with specifications, requirements and user involvement.

- The more chaotic the requirements and specification, the less the developers know what they should be building.

Defective Factors	%
Lack of User Involvement	*13*
Incomplete Requirements & Specs	*12*
Changing Requirements & Specs	*12*
Lack of Executive Support	8
Technology Incompetence	7
Lack of Resources	6
Unrealistic Expectations	*6*
Unclear Objectives	*5*
Unrealistic Time Frames	*4*
New Technology	4
Other	23

Failed Project Factors

The table lists the factors that were identified as contributing projects being canceled as ranked by people on the project

- Again, we see that requirements and specification issues and user involvement are right at the top.

- The unrealistic expectations suggests that what the business wanted to be built was beyond the technical solutions that could be reasonably be delivered by the developers.

| Failure Factors	%
Incomplete Requirements	*13*
Lack of User Involvement	*12*
Lack of Resources	11
Unrealistic Expectations	*10*
Changing Requirements & Specs	*9*
Lack of Management Support	9
Lack of Planning	*8*
Didn't Need It Any Longer	*8*
Lack of IT Management	6
Technological Illiteracy	4
Other	10

Over Engineering

The diagram shows the frequency of use of the features in a delivered software project.

- These numbers are from the Chaos report mentioned earlier.

- Only 20% of the delivered features are always or often used.

- Almost half the features are never used on average.

This means that about 45% of the development effort is wasted.

- We used resources and time to build things that never needed to be built.

- Usually happens because the feature choice is made by developers working without knowing the requirements the system needs to satisfy.

Pie chart: Always Used 7%, Often Used 13%, Sometimes Used 19%, Rarely Used 19%, Never Used 45%
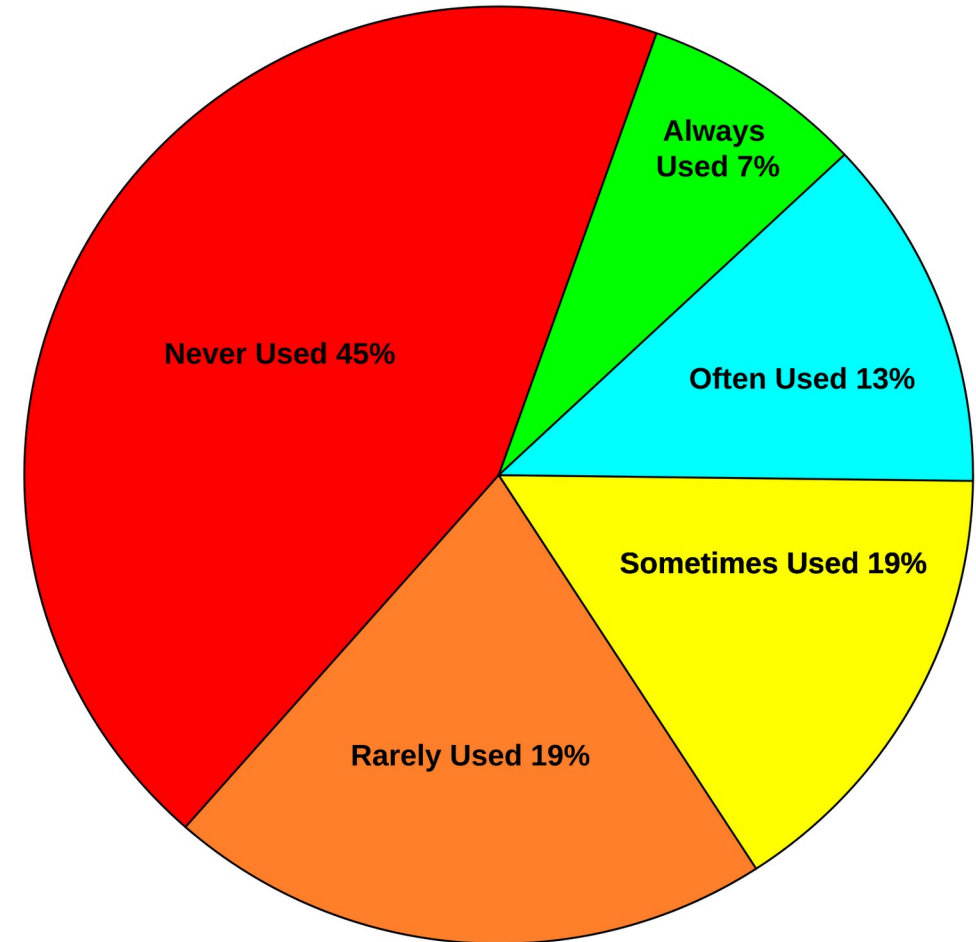
Over Engineering

Over engineering is a combination of development ineffectiveness and inefficiency.

- Because we do not know what the end user really needs, we waste resources building things that never needed to be built at all.

- The application is ineffective in solving the original motivating need or requirement.

- In terms of efficiency, if we focus on the most important features first and defer building features that contribute little value, we have increased the efficiency of the process without compromising effectiveness.
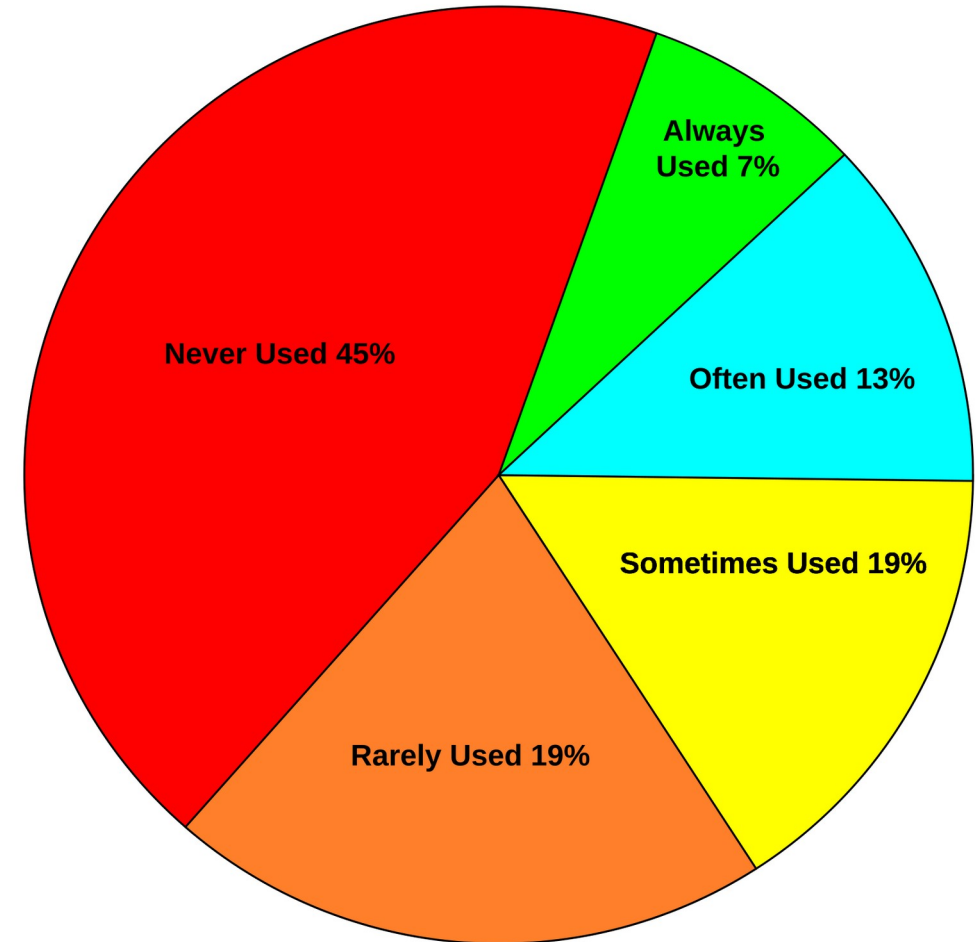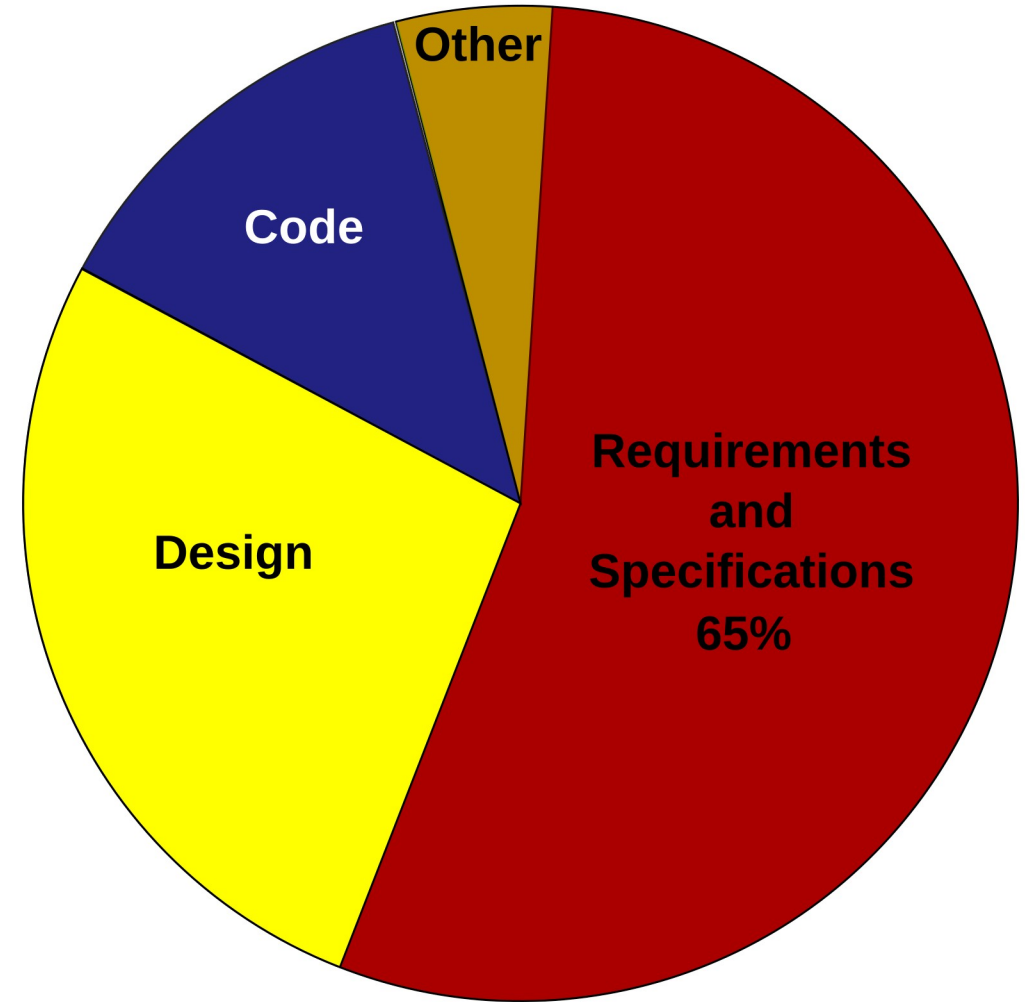
Source of Errors

In a study by Gupta et al, the place in the process where errors originated from is shown in the pie chart

What is the impact of errors in previous phases of a project?

- We either get a defective or canceled project

- Or we have to do a lot of rework to correct errors that have resulted working from wrong requirements or specification.

Rework Effort

All of this so far can seem bit academic until we ask the really important question: how much of developer time is taken up with fixing errors that could have been prevented?   :

- 61% of the IT professionals surveyed said they spent all or most of their time fixing things they felt were preventable.

- When we add in those who spend half their time fixing mistakes, then 80% of IT professionals spend at least half their time on is fixing mistakes that could have been preventU+0065d.

Never <1%

Rarely 6%

Occasionally 15%

All 21%

Half 19%

Most 40%
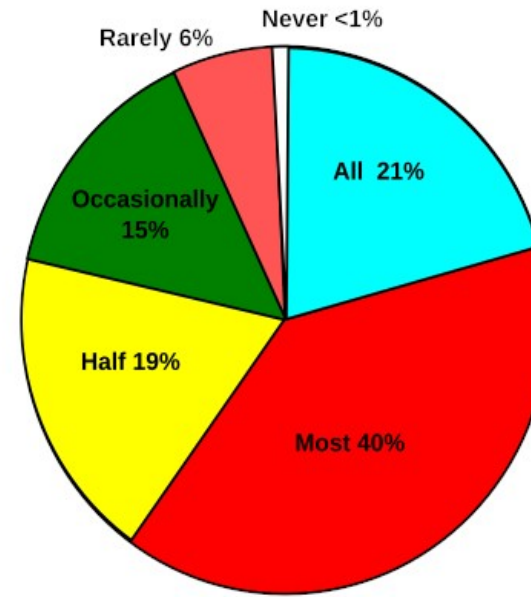
Geneca Survey of IT Professionals:

"How much of your time do you spend on avoidable or preventable rework (eg. changing requirements, missing features)?"
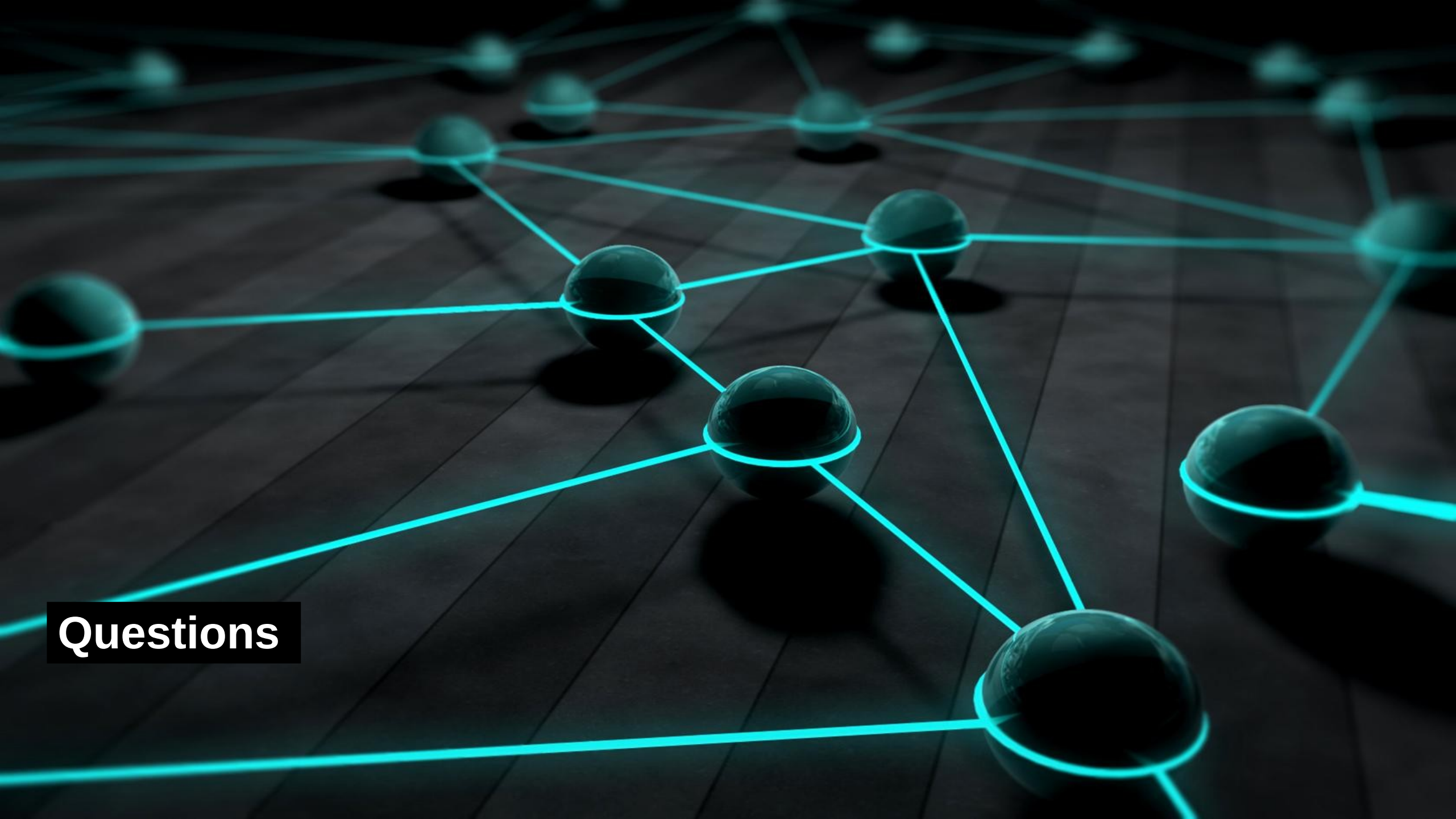
Results are consistent with other studies.

Software Engineering

- Not all programming is software engineering, nor should it be

    - Software engineering is critical for successfully delivering software when operating under time, budget and specification constraints.

    - Or in the words of Donald Knuth *"First solve the problem, THEN write the code."*

- According to The IEEE Joint Task Force on Computing Curricula; the following characterize software engineering as opposed to programming in general

    - Problem Solving
        - *Engineers proceed by making a series of decisions, carefully evaluating options, and choosing an approach at each decision-point that is appropriate for the current task in the current context. Appropriateness can be judged by trade-off analysis, which balances costs against benefit*

    - Quantitative Approach
        - *Engineers measure things, and when appropriate, work quantitatively; they calibrate and validate their measurements; and they use approximations based on experience and empirical data.*

Software Engineering

- Disciplined Process
  - *Engineers emphasize the use of a disciplined process when creating a design and can operate effectively as part of a team in doing so.*

- Tool Oriented
  - *Engineers use tools to apply processes systematically. Therefore, the choice and use of appropriate tools is key to engineering.*

- Standards and Best Practices
  - *Engineers, via their professional societies, advance by the development and validation of principles, standards, and best practices.*

- Design Reuse
  - *Engineers reuse designs and design artifacts.*

- This is not a comprehensive list but is intended to give an idea of some of the ways that software engineering is more than just programming.

Questions